Índice

1. Introducción

En el desarrollo de aplicaciones orientadas a objetos, es fundamental comprender cómo se estructuran las clases, cómo se relacionan entre sí y cómo se gestionan los errores que pueden surgir durante la ejecución. El archivo main.dart presenta una aplicación de consola escrita en Dart (usando Flutter como entorno base), que simula el funcionamiento de un sistema para la gestión de servicios en un taller mecánico. Esta aplicación incluye una jerarquía de clases que modela distintos tipos de vehículos (autos, motos y camiones), cada uno con sus propias características y lógica de cálculo de servicio, además de una interfaz común que garantiza la coherencia funcional.

1.1. Planteamiento del problema

La comprensión y mantenimiento del código puede volverse complejo si no se cuenta con una representación clara de su arquitectura y comportamiento. Además, el manejo de errores mediante excepciones está presente en varios puntos críticos del sistema, pero requiere ser analizado para entender su impacto en la robustez de la aplicación. Por ello, se realizara un reporte técnico que documente y analice los aspectos estructurales y dinámicos del sistema, así como el uso de excepciones, con el fin de mejorar la comprensión del diseño.

1.2. Motivación

Este ejercicio tiene como propósito fortalecer las habilidades de modelado y análisis en programación orientada a objetos, utilizando herramientas como los diagramas UML para representar tanto la estructura estática (clases, atributos, relaciones) como el comportamiento dinámico (interacciones, flujos de ejecución) del sistema. Además, se busca reflexionar sobre el uso de excepciones como mecanismo de control de errores, evaluando su implementación en el código y su contribución a la estabilidad del programa. Finalmente, se exige que la aplicación esté correctamente empaquetada bajo el nombre calificado mx.unam.fi.poo.p910.

1.3. Objetivos

- Diagramas UML al menos uno estático y uno dinámico.
- Dar una interpretación de la aplicación del tema de excepciones dentro del código.
- La aplicación debe estar empaquetada con el Full Qualified Name mx.unam.fi.poo.p910.

2. Marco Teórico

2.1. Clases y objetos

Dart es un lenguaje orientado a objetos con clases y herencia basada en mixins. Cada objeto es una instancia de una clase, y todas las clases, excepto Null, descienden de Object. La herencia basada en mixins significa que, aunque cada clase (excepto la clase superior, ¿Object?) tiene exactamente una superclase, el cuerpo de una clase se puede reutilizar en múltiples jerarquías de clases. Los métodos de extensión son una forma de añadir funcionalidad a una clase sin cambiarla ni crear una subclase. Los modificadores de clase permiten controlar cómo las bibliotecas pueden subtipificar una clase.[**?**]

En la programación orientada a objetos , un objeto es una unidad autocontenida de código y datos. Los objetos se crean a partir de plantillas llamadas clases. Un objeto se compone de propiedades (variables) y métodos (funciones). Un objeto es una instancia de una clase.[**?**]

- Utilizando miembros de clase:

  Los objetos tienen miembros que consisten en funciones y datos (métodos y variables de instancia , respectivamente). Al llamar a un método, se invoca sobre un objeto: el método tiene acceso a las funciones y datos de ese objeto. Utilice un punto (.) para referirse a una variable de instancia o método:

```
var p = Point(2, 2);

// Get the value of y.
assert(p.y == 2);

// Invoke distanceTo() on p.
double distance = p.distanceTo(Point(4, 4));
```

  Utilice ?.en su lugar .para evitar una excepción cuando el operando más a la izquierda sea nulo:

```
// If p is non-null, set a variable equal to its y value.
var a = p?.y;
```

- Utilizando constructores Puedes crear un objeto usando un constructor . Los nombres de los constructores pueden ser 'init' ClassNameo 'init' . Por ejemplo, el siguiente código crea objetos usando los constructores 'init' y 'init': ClassName.identifierPointPoint()Point.fromJson().[**?**]

```
var p1 = Point(2, 2);
var p2 = Point.fromJson({'x': 1, 'y': 2});
```

  El siguiente código tiene el mismo efecto, pero utiliza la newpalabra clave opcional antes del nombre del constructor:

```
var p1 = new Point(2, 2);
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

- Variables y métodos de clase

  utilice la static palabra clave para implementar variables y métodos de toda la clase.

```
class Queue {
  static const initialCapacity = 16;

}

void main() {
  assert(Queue.initialCapacity == 16);
}
```

- métodos estáticos

  Los métodos estáticos (métodos de clase) no operan sobre una instancia y, por lo tanto, no tienen acceso a ella this. Sin embargo, sí tienen acceso a las variables estáticas. Como muestra el siguiente ejemplo, los métodos estáticos se invocan directamente en una clase.[**?**]

```
import 'dart:math';

class Point {
  double x, y;
  Point(this.x, this.y);

  static double distanceBetween(Point a, Point b) {
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}

void main() {
  var a = Point(2, 2);
  var b = Point(4, 4);
  var distance = Point.distanceBetween(a, b);
  assert(2.8 < distance && distance < 2.9);
  print(distance);
}
```

2.2. Constructores

Los constructores son funciones especiales que crean instancias de clases.

Dart implementa muchos tipos de constructores. Excepto los constructores por defecto, estas funciones utilizan el mismo nombre que su clase.[**?**]

1. constructores generativos Para instanciar una clase, utilice un constructor generativo.

```
class Point {
  // Instance variables to hold the coordinates of the point.
  double x;
  double y;

  // Generative constructor with initializing formal parameters:
  Point(this.x, this.y);
}
```

2. Constructores por defecto

   Si no declaras un constructor, Dart utiliza el constructor predeterminado. El constructor predeterminado es un constructor generativo sin argumentos ni nombre.[**?**]

3. Constructores con nombre

   Utilice un constructor con nombre para implementar múltiples constructores para una clase o para proporcionar mayor claridad:

```
vconst double xOrigin = 0;
const double yOrigin = 0;
```

```
3
4  class Point {
5    final double x;
6    final double y;
7
8    // Sets the x and y instance variables
9    // before the constructor body runs.
10   Point(this.x, this.y);
11
12   // Named constructor
13   Point.origin() : x = xOrigin, y = yOrigin;
14 }
```

Una subclase no hereda el constructor con nombre de la superclase. Para crear una subclase con un constructor con nombre definido en la superclase, implemente dicho constructor en la subclase.

4. Constructores constantes

   Si tu clase produce objetos que no cambian, haz que estos objetos sean constantes en tiempo de compilación. Para ello, define un constconstructor con todas las variables de instancia establecidas como final.[**?**]

```
1  class ImmutablePoint {
2    static const ImmutablePoint origin = ImmutablePoint(0, 0);
3
4    final double x, y;
5
6    const ImmutablePoint(this.x, this.y);
7  }
```

   Los constructores constantes no siempre crean constantes. Pueden invocarse fuera de un constcontexto. Para obtener más información, consulte la sección sobre el uso de constructores .

5. Constructores de redireccionamiento

   Un constructor puede redirigir a otro constructor de la misma clase. Un constructor que redirige tiene un cuerpo vacío. El constructor utiliza thisen lugar del nombre de la clase después de dos puntos (:).

```
1  class Point {
2    double x, y;
3
4    // The main constructor for this class.
5    Point(this.x, this.y);
6
7    // Delegates to the main constructor.
8    Point.alongXAxis(double x) : this(x, 0);
9  }
```

2.3. Encapsulacion, Getters y Setters

En Dart, la encapsulación consiste en ocultar datos dentro de una biblioteca, protegiéndolos de factores externos. Esto ayuda a controlar el programa y evita que se vuelva demasiado complejo.[?]

La encapsulación se puede lograr mediante:

- Declarar las propiedades de la clase como privadas utilizando guion bajo () .
- Proporcionar métodos públicos getter y setter para acceder y actualizar el valor de la propiedad privada.

Nota: Dart no admite palabras clave como 'public' , 'private' y 'protected. Dart utiliza "'_" (guion bajo) para indicar que una propiedad o método es privado. Los métodos getter y setter se utilizan para acceder y actualizar el valor de una propiedad privada. Los métodos getter se utilizan para acceder al valor de una propiedad privada. Los métodos setter se utilizan para actualizar el valor de una propiedad privada.[?]

Ejemplo 1: En este ejemplo, crearemos una clase llamada `Employee`. Esta clase tendrá dos propiedades privadas: _id y _name. También crearemos dos métodos públicos, `getId()` y `getName()`, para acceder a las propiedades privadas. Asimismo, crearemos dos métodos públicos, `setId()` y `setName()`, para actualizar las propiedades privadas.[?]

```
1  class Employee {
2    // Private properties
3    int? _id;
4    String? _name;
5
6  // Getter method to access private property _id
7    int getId() {
8      return _id!;
9    }
10 // Getter method to access private property _name
11   String getName() {
12     return _name!;
13   }
14 // Setter method to update private property _id
15   void setId(int id) {
16     this._id = id;
17   }
18 // Setter method to update private property _name
19   void setName(String name) {
20     this._name = name;
21   }
22
23 }
24
25 void main() {
26   // Create an object of Employee class
27   Employee emp = new Employee();
28   // setting values to the object using setter
29   emp.setId(1);
30   emp.setName("John");
```

```
31
32    // Retrieve the values of the object using getter
33    print("Id: ${emp.getId()}");
34    print("Name: ${emp.getName()}");
35  }
```

En Dart, puedes controlar el acceso a las propiedades e implementar la encapsulación mediante el uso de propiedades de solo lectura. Para ello, añade la palabra clave ‘ final‘ antes de la declaración de la propiedad. De esta forma, solo podrás acceder a su valor, pero no modificarlo.[**?**]

```
1  class Student {
2    final _schoolname = "ABC School";
3
4    String getSchoolName() {
5      return _schoolname;
6    }
7  }
8
9  void main() {
10   var student = Student();
11   print(student.getSchoolName());
12   // This is not possible
13   //student._schoolname = "XYZ School";
14 }
```

Puedes crear métodos getter y setter usando las palabras clave `get` y `set`. En el siguiente ejemplo, hemos creado una clase llamada `Vehicle`. Esta clase tiene dos propiedades privadas: _model y _year. También hemos creado dos métodos getter y setter para cada propiedad. Estos métodos se llaman `model` y `year`, y se utilizan para acceder y actualizar el valor de las propiedades privadas.[**?**]

```
1  class Vehicle {
2    String _model;
3    int _year;
4
5    // Getter method
6    String get model => _model;
7
8    // Setter method
9    set model(String model) => _model = model;
10
11   // Getter method
12   int get year => _year;
13
14   // Setter method
15   set year(int year) => _year = year;
16 }
17
18 void main() {
19   var vehicle = Vehicle();
```

```
20    vehicle.model = "Toyota";
21    vehicle.year = 2019;
22    print(vehicle.model);
23    print(vehicle.year);
24  }
```

¿Por qué es importante la encapsulación?

1. Ocultación de datos : La encapsulación oculta los datos del exterior. Impide que el código fuera de la clase acceda a ellos. Esto se conoce como ocultación de datos.[**?**]
2. Facilidad de prueba : La encapsulación permite probar la clase de forma aislada. Esto permite probar la clase sin necesidad de probar el código fuera de ella.[**?**]
3. Flexibilidad : La encapsulación permite cambiar la implementación de la clase sin afectar al código que se encuentra fuera de la clase.[**?**]
4. Seguridad : La encapsulación permite restringir el acceso a los miembros de la clase. Esto permite limitar el acceso a los miembros de la clase desde el código externo a la biblioteca.[**?**]

2.4. Herencia

La herencia es la compartición de comportamiento entre dos clases. Permite definir una clase que extiende la funcionalidad de otra. La palabra clave 'extend' se utiliza para heredar de la clase padre.[**?**]

Cuando se utiliza la herencia, siempre se crea una relación .^{es} un/una.^{en}tre la clase padre y la clase hija, como por ejemplo: Estudiante es una Persona , Camión es un Vehículo , Vaca es un Animal , etc.

Dart admite herencia simple, lo que significa que una clase solo puede heredar de una única clase. Dart no admite herencia múltiple, lo que significa que una clase no puede heredar de varias clases.[**?**]

```
1  class ParentClass {
2    // Parent class code
3  }
4
5  class ChildClass extends ParentClass {
6    // Child class code
7  }
```

En esta sintaxis, ParentClass es la superclase y ChildClass es la subclase. ChildClass hereda las propiedades y métodos de ParentClass .

Términos:

- Clase padre: La clase cuyas propiedades y métodos hereda otra clase se denomina clase padre. También se la conoce como clase base o superclase.[**?**]
- Clase hija: La clase que hereda las propiedades y métodos de otra clase se denomina clase hija. También se la conoce como clase derivada o subclase.[**?**]

Ejemplo 1: En este ejemplo, crearemos una clase Persona y luego crearemos una clase Estudiante que hereda las propiedades y métodos de la clase Persona

```
class Person {
  // Properties
  String? name;
  int? age;

  // Method
  void display() {
    print("Name: $name");
    print("Age: $age");
  }
}
// Here In student class, we are extending the
// properties and methods of the Person class
class Student extends Person {
  // Fields
  String? schoolName;
  String? schoolAddress;

  // Method
  void displaySchoolInfo() {
    print("School Name: $schoolName");
    print("School Address: $schoolAddress");
  }
}

void main() {
  // Creating an object of the Student class
  var student = Student();
  student.name = "John";
  student.age = 20;
  student.schoolName = "ABC School";
  student.schoolAddress = "New York";
  student.display();
  student.displaySchoolInfo();
}
```

Ventajas de la herencia:

- Promueve la reutilización del código y reduce el código redundante
- Ayuda a diseñar un programa de mejor manera.
- Simplifica y limpia el código, y ahorra tiempo y dinero en mantenimiento.
- Facilita la creación de bibliotecas de clases.
- Puede utilizarse para imponer una interfaz estándar a todas las clases hijas.

2.5. Polimorfismo

Poli significa muchos y morfo significa formas . El polimorfismo es la capacidad de un objeto para adoptar muchas formas. Como seres humanos, tenemos la capacidad de adoptar muchas formas.

Podemos ser estudiantes, profesores, padres, amigos, etc. De manera similar, en la programación orientada a objetos, el polimorfismo es la capacidad de un objeto para adoptar muchas formas.[?]

La sobreescritura de métodos es una técnica que permite crear un método en la clase hija con el mismo nombre que el método en la clase padre. El método en la clase hija sobreescribe el método en la clase padre.[?]

```
class ParentClass{
void functionName(){
  }
}
class ChildClass extends ParentClass{
@override
void functionName(){
  }
}
```

Ejemplo 1: Polimorfismo mediante la sobreescritura de métodos en Dart

En el siguiente ejemplo, existe una clase llamada Animal con un método llamado eat() . El método eat() se redefine en la clase hija llamada Dog .

```
class Animal {
  void eat() {
    print("Animal is eating");
  }
}

class Dog extends Animal {
  @override
  void eat() {
    print("Dog is eating");
  }
}

void main() {
  Animal animal = Animal();
  animal.eat();

  Dog dog = Dog();
  dog.eat();
}
```

Ventaja de usar polimorfismo:

1. Las subclases pueden anular el comportamiento de la clase padre.
2. Nos permite escribir código más flexible y reutilizable.

2.6. Clase abstracta

Las clases abstractas son clases que no se pueden inicializar. Se utilizan para definir el comportamiento de una clase que puede ser heredado por otras clases. Una clase abstracta se declara

utilizando la palabra clave abstract.[?]

```
abstract class ClassName {
  //Body of abstract class

  method1();
  method2();
}
```

Un método abstracto es un método que se declara sin implementación. Se declara con un punto y coma (;) en lugar del cuerpo del método.[?]

```
abstract class ClassName {
  //Body of abstract class
  method1();
  method2();
}
```

Las subclases de una clase abstracta deben implementar todos los métodos abstractos de la clase abstracta. Se utiliza para lograr la abstracción en el lenguaje de programación Dart.[?]

Ejemplo 1: En el siguiente ejemplo, hay una clase abstracta Vehículo con dos métodos abstractos start() y stop(). Las subclases Auto y Bicicleta implementan los métodos abstractos y los sobrescriben para imprimir el mensaje..[?]

```
abstract class Vehicle {
  // Abstract method
  void start();
  // Abstract method
  void stop();
}

class Car extends Vehicle {
  // Implementation of start()
  @override
  void start() {
    print('Car started');
  }

  // Implementation of stop()
  @override
  void stop() {
    print('Car stopped');
  }
}

class Bike extends Vehicle {
  // Implementation of start()
  @override
  void start() {
```

```
26      print('Bike started');
27    }
28
29    // Implementation of stop()
30    @override
31    void stop() {
32      print('Bike stopped');
33    }
34  }
35
36  void main() {
37    Car car = Car();
38    car.start();
39    car.stop();
40
41    Bike bike = Bike();
42    bike.start();
43    bike.stop();
44  }
```

Nota: La clase abstracta se utiliza para definir el comportamiento de una clase que puede ser heredada por otras clases. Se puede definir un método abstracto dentro de una clase abstracta.[**?**]

No se puede crear un objeto de una clase abstracta. Sin embargo, se puede definir un constructor en una clase abstracta. El constructor de una clase abstracta se invoca cuando se crea un objeto de una subclase.[**?**]

Ejemplo 2: En el siguiente ejemplo, hay una clase abstracta Bank con un constructor que toma dos parámetros: nombre y tasa. Hay un método abstracto interest(). Las subclases SBI e ICICI implementan el método abstracto y lo sobrescriben para imprimir la tasa de interés.

```
1   abstract class Bank {
2     String name;
3     double rate;
4
5     // Constructor
6     Bank(this.name, this.rate);
7
8     // Abstract method
9     void interest();
10
11    //Non-Abstract method: It have an implementation
12    void display() {
13      print('Bank Name: $name');
14    }
15  }
16
17  class SBI extends Bank {
18    // Constructor
19    SBI(String name, double rate) : super(name, rate);
20
```

```
21    // Implementation of interest()
22    @override
23    void interest() {
24      print('The rate of interest of SBI is $rate');
25    }
26  }
27
28  class ICICI extends Bank {
29    // Constructor
30    ICICI(String name, double rate) : super(name, rate);
31
32    // Implementation of interest()
33    @override
34    void interest() {
35      print('The rate of interest of ICICI is $rate');
36    }
37  }
38
39  void main() {
40    SBI sbi = SBI('SBI', 8.4);
41    ICICI icici = ICICI('ICICI', 7.3);
42
43    sbi.interest();
44    icici.interest();
45    icici.display();
46  }
```

3. Desarrollo

A continuacion se mostraran primero los diagrmas UML, comenzando con el estatico.

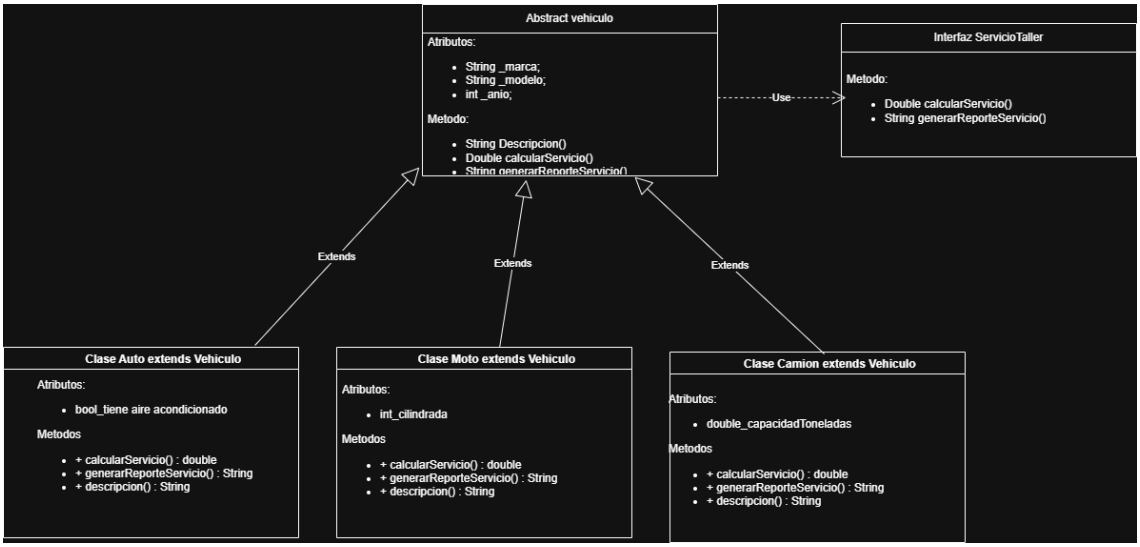


Figura 1: Diagrama estatico

El diseño de clases propuesto implementa un sistema para gestionar servicios de vehículos de un taller mecánico, aplicando los pilares fundamentales de la Programación Orientada a Objetos (POO): Abstracción, Herencia, y Polimorfismo.

1. Interfaz (ServicioTaller) y Contrato

El diseño se inicia con la Interfaz ServicioTaller, la cual define un contrato de funcionalidad que debe ser implementado obligatoriamente por cualquier clase que quiera ser gestionada como un ”Servicio de Taller”.

- Métodos del Contrato:
  - calcularServicio(): Obliga a calcular el costo del servicio.
  - generarReporteServicio(): Obliga a generar un informe detallado del servicio.

Esta interfaz garantiza la interoperabilidad: cualquier vehículo que cumpla con este contrato puede ser tratado de forma genérica como un objeto capaz de ser atendido en el taller.

2. Clase Base Abstracta (Vehiculo) y Abstracción

La clase Vehiculo es una clase abstracta. Esto significa que no se pueden crear instancias directas de Vehiculo; solo se pueden instanciar sus subclases concretas.

- Implementación de Interfaz: La clase Vehiculo es la que formalmente implementa la interfaz ServicioTaller. Esto asegura que todos los vehículos, independientemente de su tipo, cumplirán con el contrato de servicio.

- Abstracción de Atributos Comunes: Esta clase encapsula los atributos esenciales compartidos por todos los vehículos: _marca, _modelo, y _anio. También proporciona métodos comunes, como descripcion(), que ofrece una representación básica del vehículo.

3. Clases Concretas y Herencia

Las clases Auto, Moto, y Camion son las clases concretas.

- Relación de Herencia (Extends): Cada una de estas clases hereda de la clase Vehiculo, adquiriendo automáticamente sus atributos y el compromiso de implementar los métodos del contrato de servicio.
- Atributos Específicos: Cada clase introduce atributos únicos relevantes para su tipo (_tieneAireAcondicionado en Auto, _cilindrada en Moto, _capacidadToneladas en Camion).

4. Polimorfismo y Especialización

El principio de Polimorfismo (múltiples formas) es fundamental en este diseño y se evidencia en la sobreescritura (@override) de los métodos heredados.

- Sobreescritura de Métodos (@override): Aunque todos los vehículos deben calcularServicio() y generarReporteServicio() (por el contrato de la interfaz), la lógica de cómo se hace es específica para cada tipo de vehículo.
- Ejemplo: El método calcularServicio() en la clase Auto incluye un recargo por aire acondicionado, mientras que en la clase Camion incluye costos por revisión de frenos y suspensión, y un recargo por capacidad de carga.

Esta especialización a través del polimorfismo permite que el sistema trate a todos los objetos de manera uniforme (como un ServicioTaller), mientras ejecuta el comportamiento adecuado para su tipo específico.

Para el diagrama dinámico:

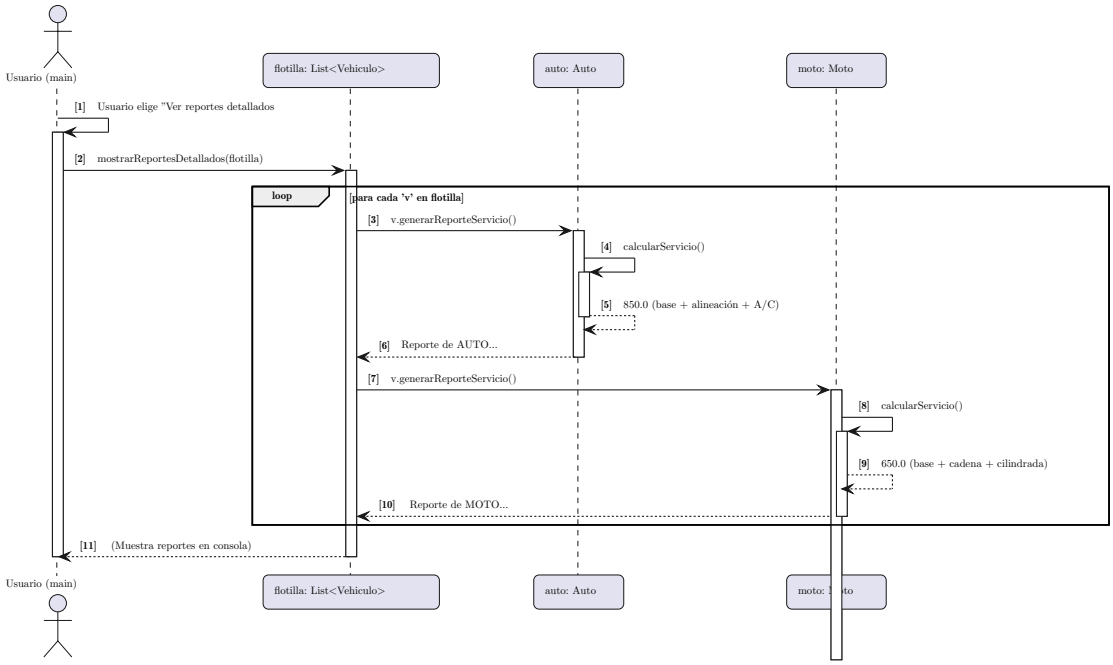


Figura 2: Diagrama de Secuencia (Dinámico).

Para cumplir con el modelado del comportamiento, se representa el polimorfismo que ocurre cuando el usuario selecciona la opción "Ver reportes detallados". El diagrama muestra cómo la función **mostrarReportesDetallados** trata a todos los vehículos disponibles de la misma manera pero cada objeto responde de manera diferente.

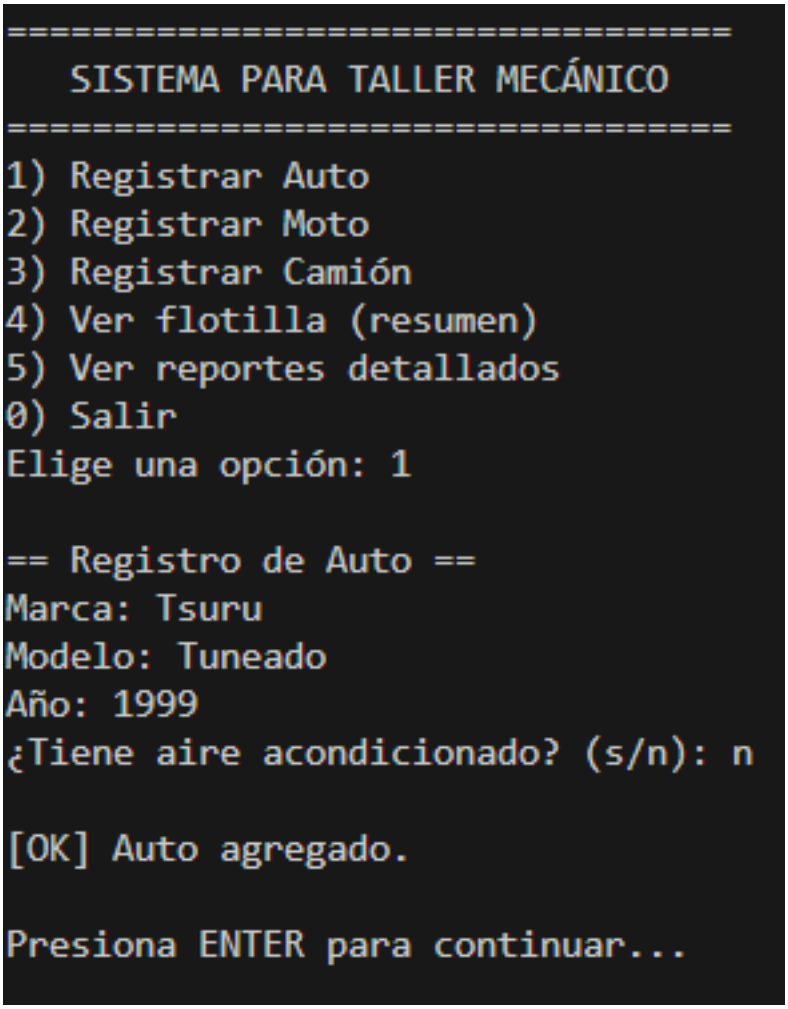
Polimorfismo La flotilla no necesita saber si se trata de un auto, motocicleta o camión, trata a todos sus elementos como un vehículo genérico. Envía el mismo mensaje -*generarReporteServicio()*- y, gracias al enlace dinámico -*dynamic binding*- de Dart, el sistema selecciona en tiempo de ejecución la versión correcta @override del método que debe ejecutarse.

Esta misma no sabe cómo auto calculó su servicio, simplemente confía en que el objeto sabe cómo hacerlo, esto se debe a la lógica interna llamada *calcularServicio()*, que está oculta dentro del objeto, es decir, está encapsulado.

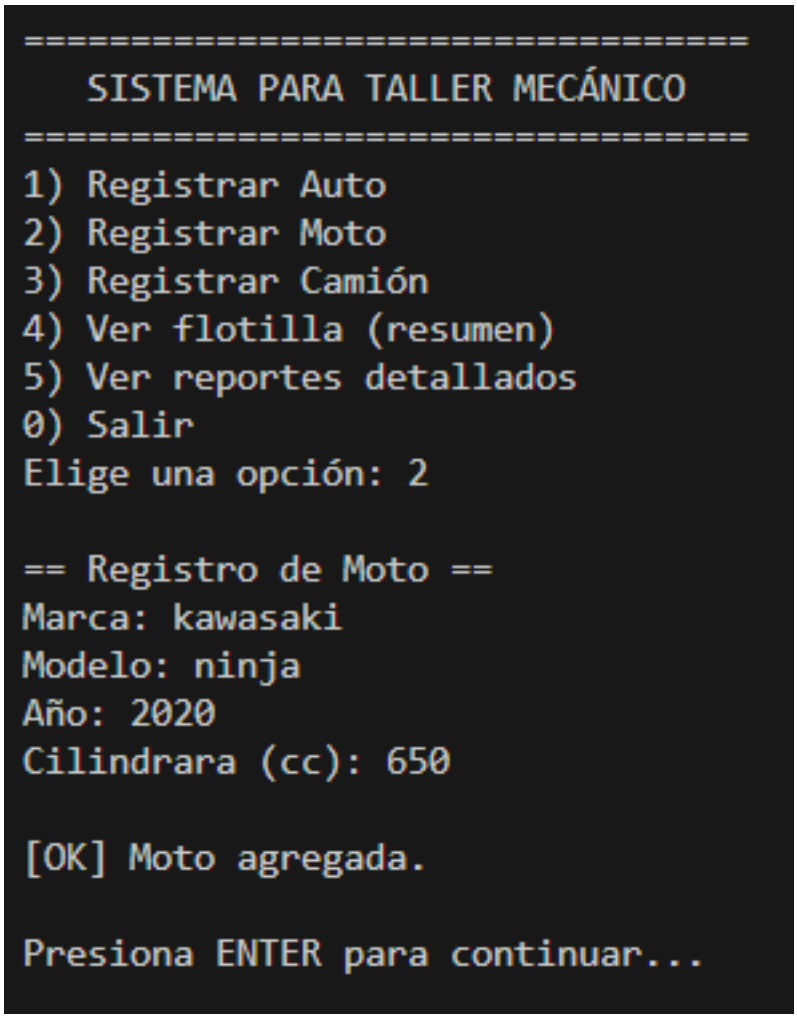
Bucle y mensaje polimórfico La función que opera sobre el objeto flotilla inicia un bucle que itera sobre cada elemento "v"de su colección. Dentro del bucle, flotilla envía el mismo mensaje -*v.generarReporteServicio()*- al primer objeto, que resulta ser un auto. Este mismo recibe el mensaje y completa el reporte haciendo una auto-llamada de *calcularServicio()* para devolver el reporte formateado.

Para la siguiente iteración el bucle envía el mismo mensaje al siguiente objeto que es una moto, hace lo mismo que auto pero hace su propia versión de *calcularServicio()* que en este caso utiliza la cilindrada para efectuar su cálculo. El bucle continuará de la misma manera para los demás vehículos, finalmente la función *mostrarReportesDetallados* imprime los resultados en la consola.

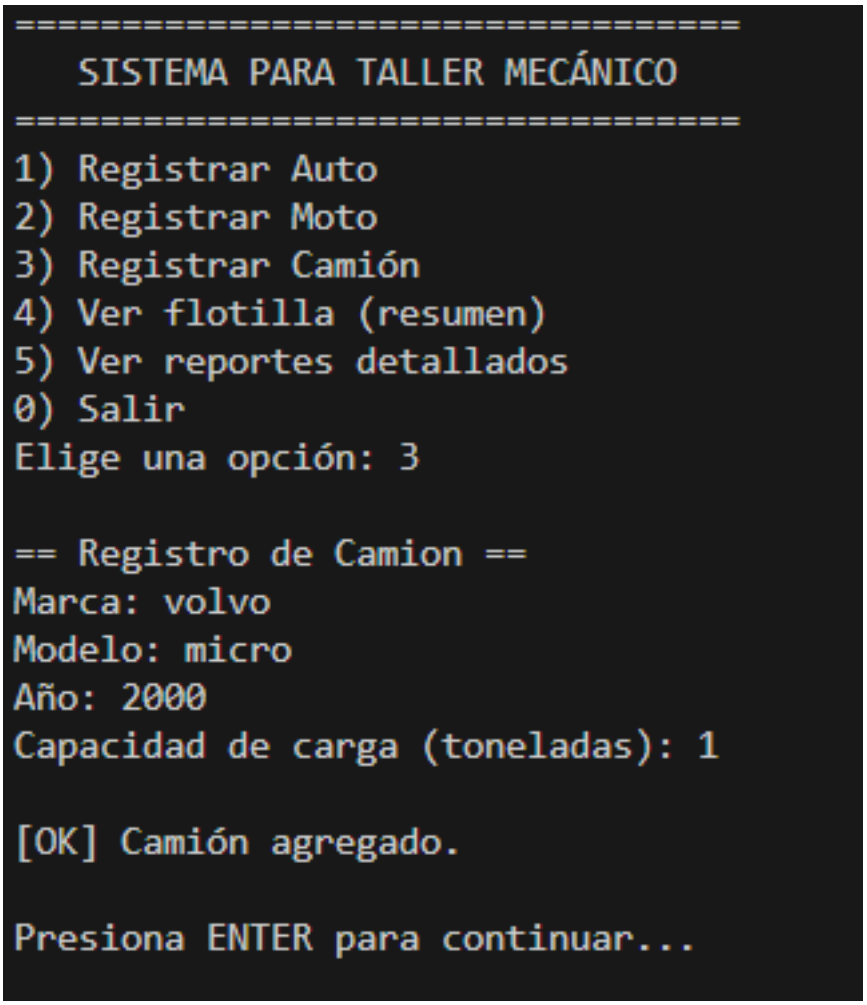
4. Resultados



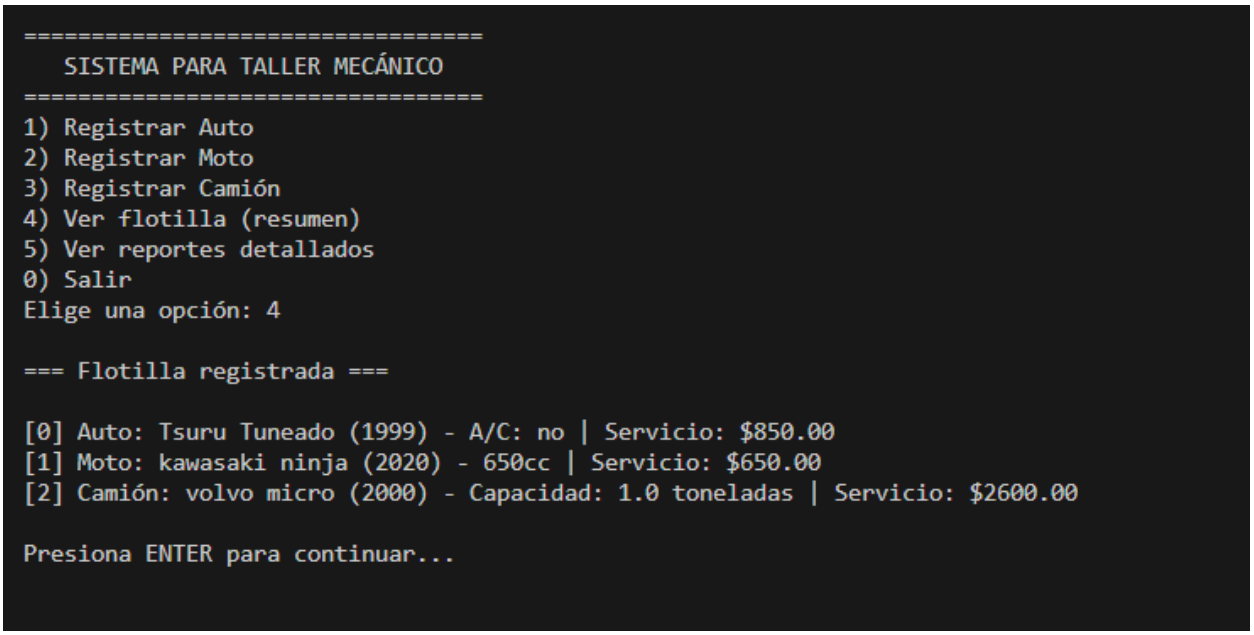
Se muestra el menú de la app, ademas de la primer función **Registrar auto"**, donde al ingresar la opción te pide ciertos datos del auto y al final te confirma que se agrego de manera correcta.



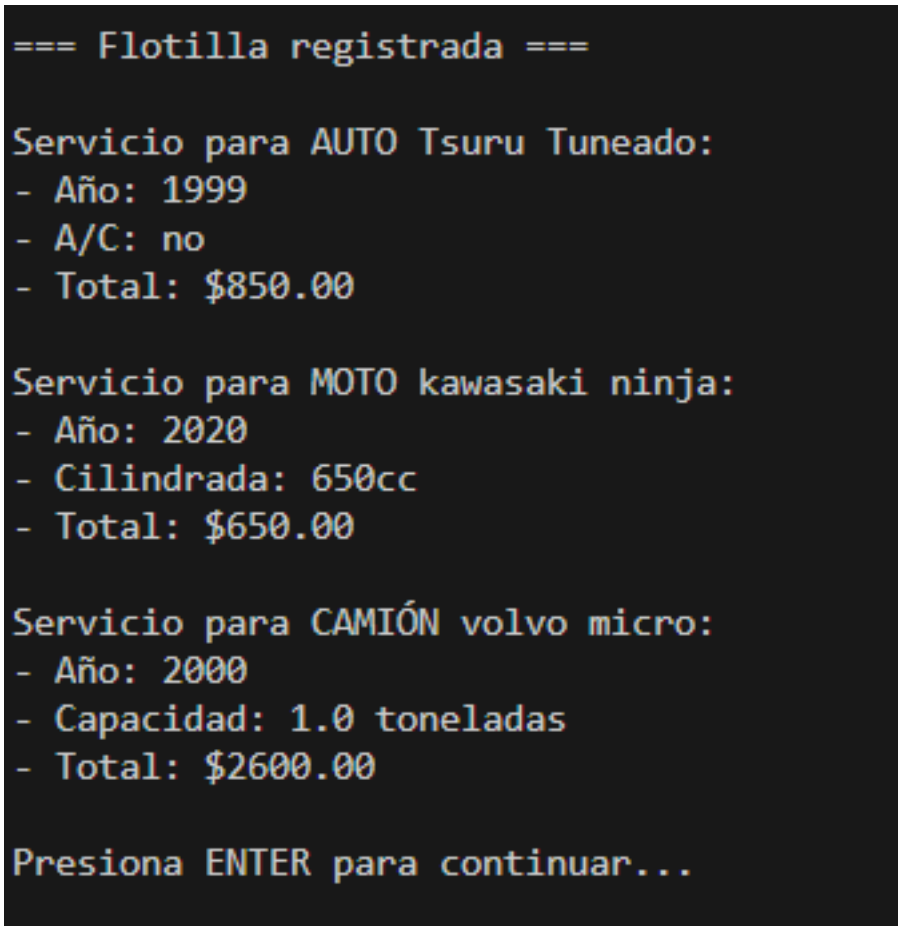
Se muestra el menú de la app, al seleccionar la segunda opción **Registrar moto"**, donde al ingresar la opción te pide ciertos datos de la moto y al final te confirma que se agrego de manera correcta.



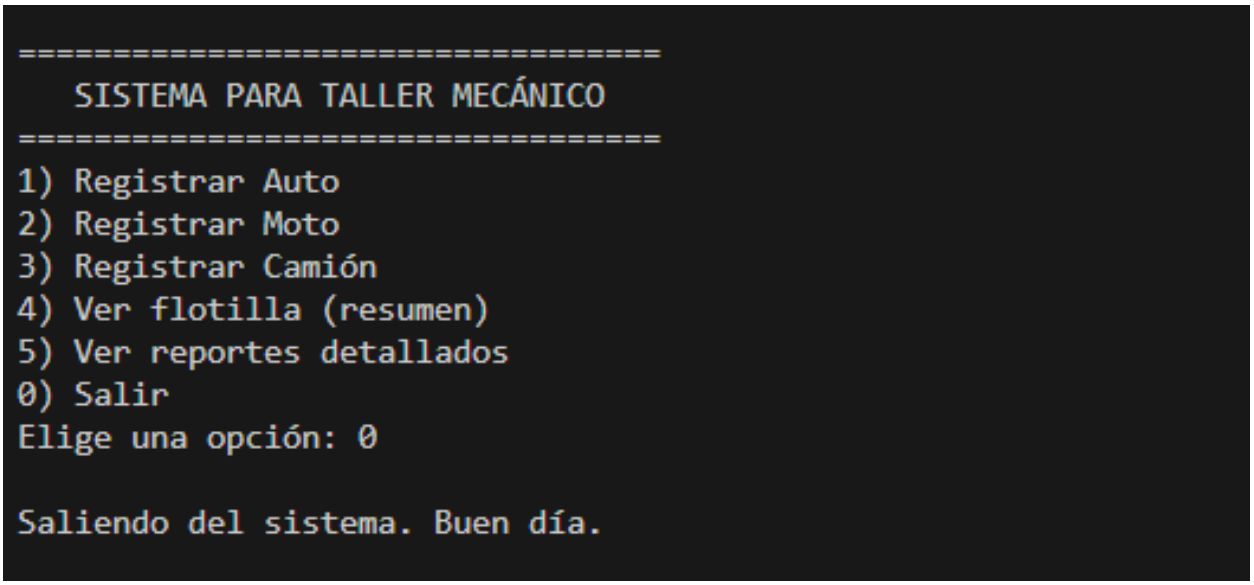
Se muestra el menú de la app, al seleccionar la tercer opción **Registrar camión"**, donde al ingresar la opción te pide ciertos datos del camión y al final te confirma que se agrego de manera correcta.



Se muestra el menú de la app, al seleccionar la cuarta opción **"Ver flotilla"**, donde al ingresar la opción te brinda los vehículos que ingresaste en el orden que los ingresaste, ademas de algunos detalles de cada vehículo y el costo de su servicio.



Se muestra el menú de la app, al seleccionar la quinta opción **"Ver reportes detallados"**, donde al ingresar la opción se muestra una lista con los autos vehículos ingresados y el costo de sus servicios.



Se muestra el menú de la app, al seleccionar ultima opción **"salir"**, se muestra una respuesta que te indica que se esta saliendo del sistema correctamente,

5. Conclusiones

La práctica realizada permitió alcanzar de manera integral los objetivos planteados, fortaleciendo la comprensión y aplicación de los principios de la programación orientada a objetos en el contexto de un sistema de gestión de servicios para un taller mecánico. Se definió un contrato claro mediante la interfaz *ServicioTaller* (con **calcularServicio()** y **generarReporteServicio()**), y se organizó la jerarquía alrededor de la clase *abstracta* **Vehiculo** y las clases concretas **Auto**, **Moto** y **Camion**. Este diseño aseguró coherencia y reutilización del comportamiento común, mientras que la herencia y la sobrescritura evidenciaron el **polimorfismo** a través de una misma interfaz de uso.

Adicionalmente, se reforzó el **encapsulamiento** de atributos mediante *getters* y *setters*, y se documentó la estructura con diagramas UML, facilitando la comunicación técnica y la mantenibilidad. La organización del código mediante el *Full Qualified Name* propuesto contribuyó a la **modularidad** y a una estructura clara del proyecto.

En conjunto, los resultados muestran que el modelo propuesto es consistente y extensible: permitió registrar distintos tipos de vehículos, calcular costos de servicio y generar reportes desde el menú de la aplicación. En síntesis, la práctica consolidó **abstracción**, **herencia**, **polimorfismo**, **encapsulación** y **modularidad** como fundamentos para escalar el sistema sin comprometer su cohesión ni su claridad.